VIA PadLock
Hash Engine
Programming Guide

Version 1.32

This is **Version 1.32** of the VIA PadLock Hash Engine Programming Guide.

LIFE SUPPORT POLICY

VIA processor products are not authorized for use as components in life support or other medical devices or systems (hereinafter called life support devices) unless a specific written agreement pertaining to such intended use is executed between the manufacturer and an officer of VIA.

1. Life support devices are devices which (a) are intended for surgical implant into the body or (b) support or sustain life and whose failure to perform, when properly used in accordance with instructions for use provided in the labeling, can be reasonably expected to result in a significant injury to the user.
2. This policy covers any component of a life support device or system whose failure to perform can cause the failure of the life support device or system, or to affect its safety or effectiveness.

Table of Contents

CHAPTER

1

INTRODUCTION

VIA provides a suite of security technologies called PadLock in all new processors, beginning with PadLock Random Number Generator in the VIA Nehemiah processor.

The VIA Esther processor adds to PadLock a number of new capabilities. This programming guide documents the VIA PadLock Hash Engine (PHE), a hardware implementation of the Secure Hash Algorithms SHA-1 and SHA-256, as specified in Federal Information Processing Standards Publication 180-2 (FIPS 180-2).

Separate Programming Guides are available for PadLock Random Number Generator (RNG), PadLock Montgomery Multiplier (PMM), and PadLock Advanced Cryptography Engine (ACE).

The CPUID identification of the VIA Esther processor is:

Vendor ID:	CentaurHauls
Type:	0
Family:	6
Model:	10

For support, developers should contact ace_support@centtech.com.

CHAPTER

2

ARCHITECTURE

2.1 HARDWARE

PadLock Hash Engine (PHE) is implemented in hardware with associated microcode. The hardware performs the hashing functions, and the microcode provides a well-designed x86 instruction interface providing transparent multitasking. This section describes the PadLock Hash Engine hardware. This information is hidden from the x86 instruction interface and is not necessary for proper use of PHE instructions.

Figure 1 illustrates the conceptual architecture of PadLock Hash Engine, a 128-bit wide unit added in parallel with the SSE unit sharing instruction, load, and store buses. PHE has a discrete set of internal registers that are not visible to x86 instructions. PHE instructions are issued, in order, to the SSE unit along with SSE instructions.

Once a PHE instruction is ready for execution, it is dispatched to the PHE hardware by the SSE unit. Instruction execution in other units, such as integer, MMX, and floating point, can proceed in parallel with PHE instructions. SSE instructions, however, are blocked from execution until the PHE instruction completes, because all PHE operations end with storing data to memory. This store must be completed before subsequent SSE instructions can execute

Figure 2 provides a further conceptual breakout of PHE components. There are three input registers and two output registers. The initial constants defined for the hash specification are loaded from memory via the SSE load bus to the INPUT_0 and the INPUT_1 registers. When the hash function uses less than 256 bits of initial constants (SHA-1 for example only needs 160 bits), the extra high-order bits in the INPUT_1 register are ignored by the hardware.

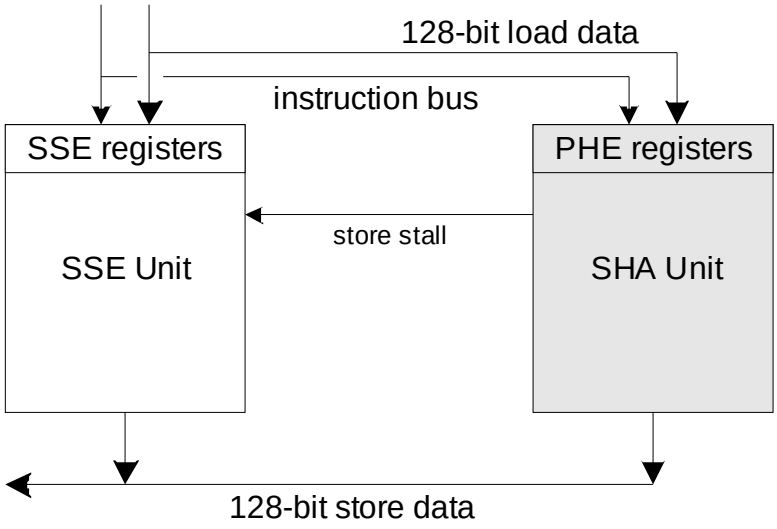
The bytes to be hashed (including any padding) are loaded, 16 bytes at a time, to the DATA register until enough bytes have been loaded to satisfy the block size of the selected hash function. SHA-1 and SHA-256 both operate on 64-byte blocks, so there are four loads from the SSE bus to the DATA register.

When the hash engine completes the specified number of rounds, the current value of the hash is placed in the OUTPUT_0 and OUTPUT_1 registers and written to memory via the SSE store bus and forwarded to the INPUT_0 and INPUT_1 registers. This result is now immediately available for the next data block(s), and

this cycle continues until the hash is complete or until an interrupt is serviced after one the store operations. Since the current value of the hash is present on memory, when the instruction restarts that hash value will be loaded in INPUT_0 and INPUT_1 and the hash engine state will be as though the interrupt did not occur.

The hardware occupies less than 0.5 mm^2 on the VIA processors and runs at the processor clock frequency.

Figure 1. Adding Security Features to x86 Architecture

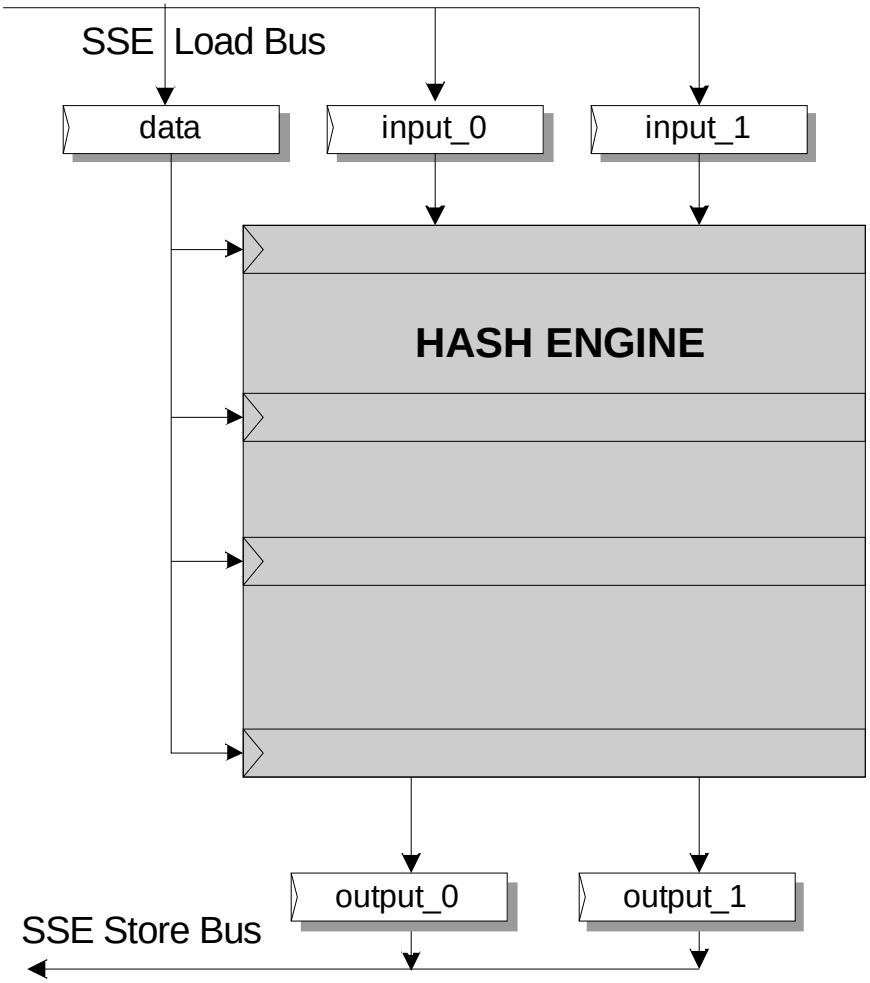


Rather than detail the hardware logic of the PadLock Hash Engine, interested readers are referred to the defining specification for the Secure Hash Standard: Federal Information Processing Standards Publication 180-2.

http://csrc.nist.gov/publications/fips/fips180-2/fips180-2.pdf

Sample code that implements SHA-1 can be obtained by looking at the source code for the random number device of many Open Source Software Operating Systems. It is also available from various websites. If you choose to use such code, perhaps to support those so unfortunate as to not have a VIA Esther processor (!), be sure to comply with the License under which that code is published.

Figure 2. Conceptual PHE Hardware Overview



2.2 SOFTWARE

PHE functions are accessed using one new x86 primary opcode. This is a group opcode providing eight sub-opcode functions. PHE instruction operands are defined by multiple memory values whose addresses are contained in General Purpose Registers. The approach is similar to that used by x86 string instructions.

The major components are:

- Two bits in the Centaur extended feature flags returned by the CPUID instruction. These bits identify (1) whether PHE physically exists on the processor and (2) whether it is currently enabled.
- A bit in the existing FCR MSR that allows a privileged program to enable or disable PHE.
- One non-privileged x86 opcode for VIA processor cryptographic features. Figure 3 illustrates the instruction architecture and opcodes, and Figure 4 illustrates the eight group instructions.

This new PHE primary opcode is the same opcode used by VIA processors for the Montgomery Multiplier. PHE defines two of the eight sub-opcodes. The remaining five sub-opcodes are **reserved** by VIA for new security features.

Figure 3. General Hash Engine Format

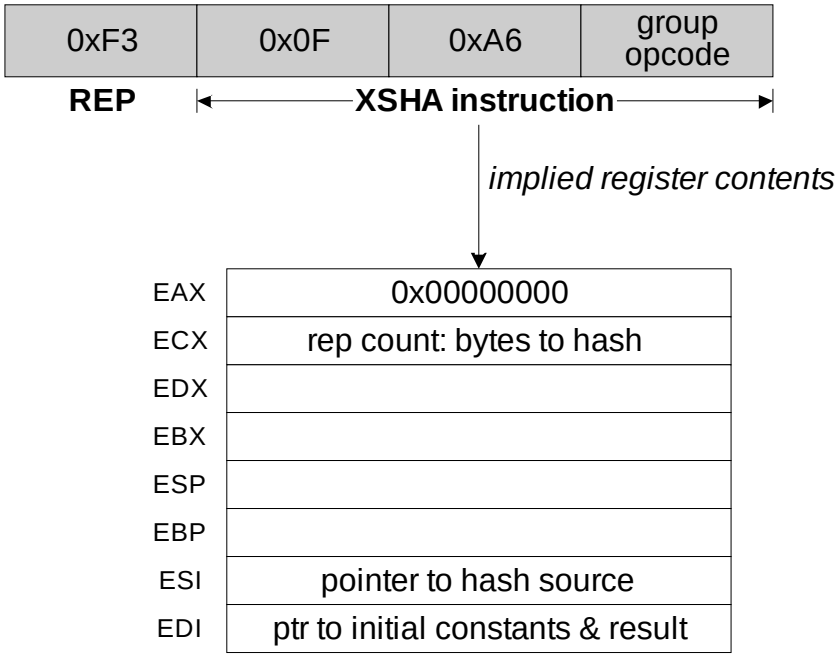
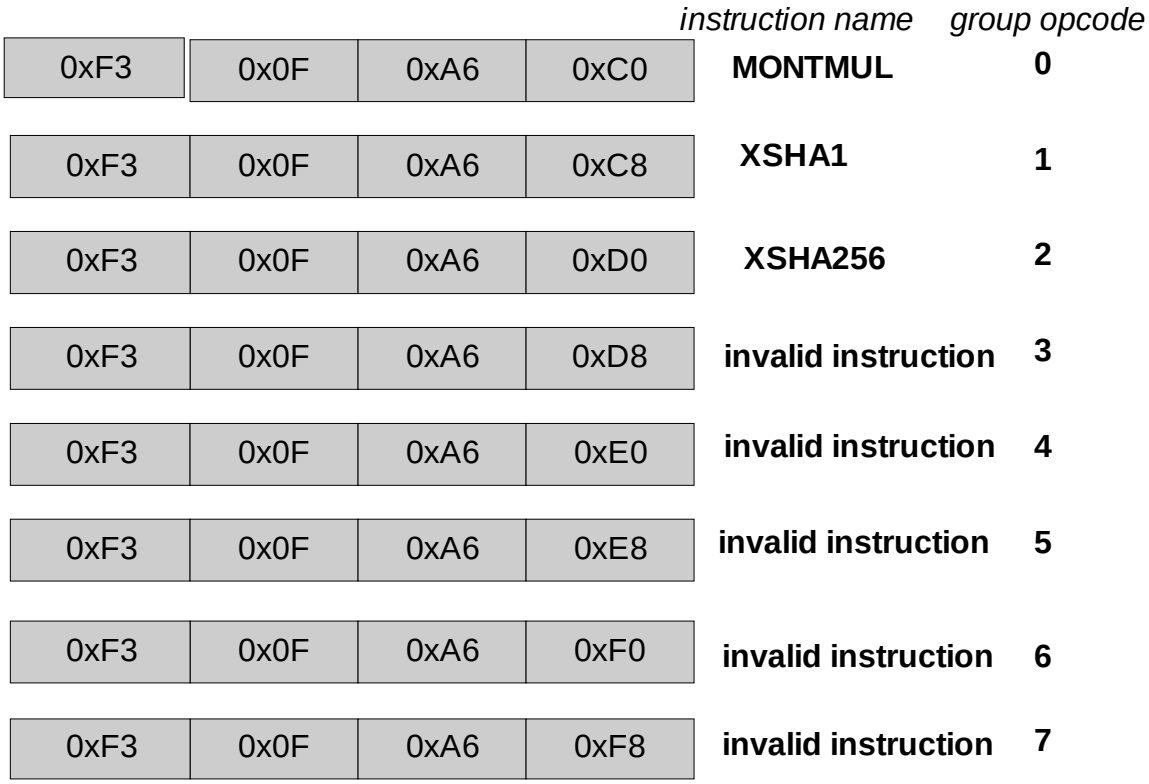


Figure 4. Instruction Functions



The XSHA instructions are atomic with respect to the hashing of a single data block (512 bits, as defined by the FIPS 180-2 specification); that is, once execution starts, the hashing function completes for that block before interrupts are allowed.

All XSHA instructions exist in REP form only. Like other REP x86 instructions, they can be interrupted and restarted. As it is necessary when performing the hash function to know whether or not one is operating on the last block of input, a non-REP form for XSHA makes no good sense.

The XSHA instructions define their operands in x86 General Purpose Registers.

2.3 MULTI-TASKING

PHE architecture is specifically designed to address two major (and related) goals:

- To allow multiple applications (including the operating system) to use XSHA instructions without any operating systems support (such as a device driver), and
- To allow multiple applications (including the operating system) to use XSHA instructions without any awareness of, or visibility of, other tasks (and their data) that are also using XSHA instructions.

That is, if

- Task A executes an XSHA instruction, and
- A task switch subsequently occurs to task B that subsequently executes an XSHA instruction, the following must be true:

Any inter-instruction state needed to later restart task A and execute subsequent task A XSHA instructions must be saved (when task A is suspended) and restored (when task A is restarted) by existing operating system code, and

Any inter-instruction state required to execute task B's XSHA instructions correctly must be restored (when task B is started) by *existing* operating system code, and

Task B must not be able to see, touch or smell anything relating to task A's use of XSHA instructions, and vice versa.

The VIA approach to satisfy these requirements is to make XSHA self-contained. That is, all of the information needed to perform a hash is contained within an XSHA instruction by using pointers in general purpose registers to all of the operands. These pointers are saved across task switches by existing operating systems.

As there is no state information within PHE that is not preserved in x86 architecture registers or in the user program memory, the xsha instructions are safe multi-tasking.

However, PHE instructions use portions of the Advanced Cryptography Engine control word logic to distinguish between 160 bit (XSHA1) and 256 bit (XHA256) hash functions. Because the ACE control word is written to by the microcode of PHE, XSHA instructions set the ACE EFLAGS:30 bit to zero to indicate that any ACE functions must reload the control word and encryption key.

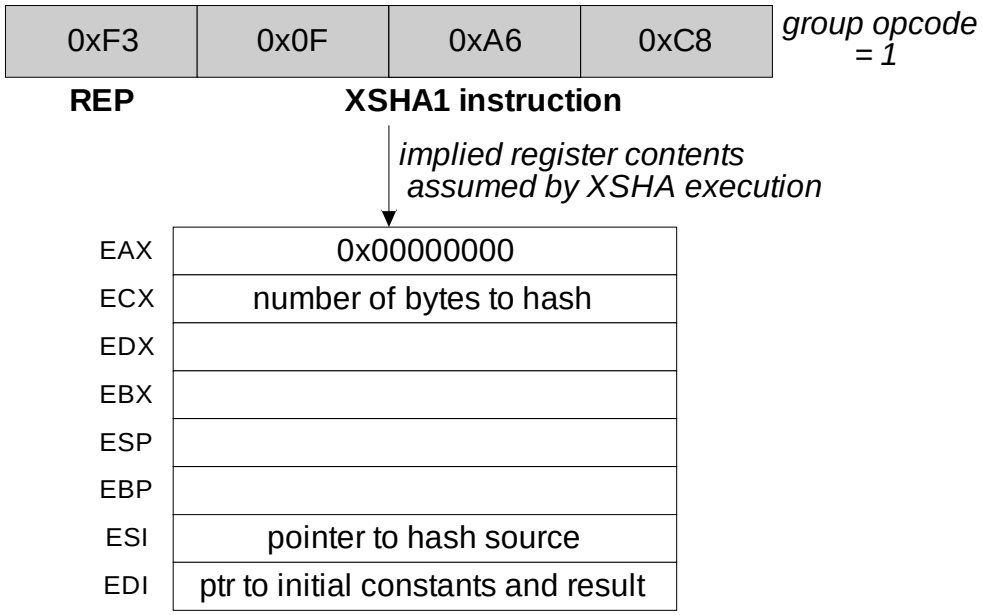
Programmers who are both encrypting and hashing a message may find it more efficient to complete the entire hash function before encrypting any blocks of the message.

2.4 XSHA1: 160 BIT HASH FUNCTION

Figure 5 shows the detailed format of the REP XSHA1 instruction. This performs the original Secure Hash Algorithm, producing a 160-bit hash of the source as defined by FIPS 180-2.

The basic steps to use the instruction: [**Note: All addresses assume the ES segment selector, which may not be overridden]**

Figure 5. XSHA1 Instruction



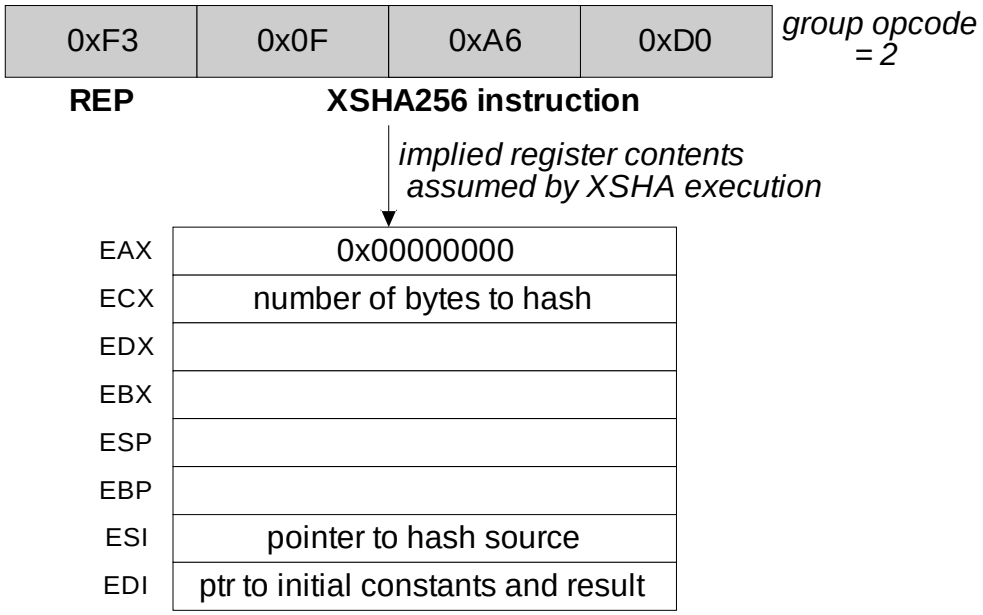
Check the Centaur extended CPUID flags to see if PHE is present.

1. If necessary, enable PHE using the FCR MSR. This is a privileged operation but rarely needs to be done, as the default state at RESET is that PHE is enabled.
2. Set EAX = 0x00000000.
3. Load the effective address (offset) of the first byte of the data to be hashed into ESI.
4. Load the effective address (offset) of the result buffer (the hash) into EDI**. This address must be aligned on a 16-byte boundary.** This buffer must also be initialized with the correct constants for SHA-1: (0x67452301, 0xefcdab89, 0x98badcfe, 0x10325476, 0xc3d2e1f0)
5. Load the number of bytes to be hashed into ECX.
6. Execute the REP XSHA1 instruction.

2.5 XSHA256: 256 BIT HASH FUNCTION

Figure 6 shows the detailed format of the REP XSHA256 instruction. This performs the SHA-256 Algorithm, producing a 256-bit hash of the source as defined by FIPS 180-2.

Figure 6 XSHA256 instruction



The basic steps to use the instruction: [**Note: All addresses assume the ES segment selector, which may not be overridden]**

7. Check the Centaur extended CPUID flags to see if PHE is present.
8. If necessary, enable PHE using the FCR MSR. This is a privileged operation but rarely needs to be done, as the default state at RESET is that PHE is enabled.
9. Set EAX = 0x00000000.
10. Load the effective address (offset) of the first byte of the data to be hashed into ESI.
11. Load the effective address (offset) of the result buffer (the hash) into EDI**. This address must be aligned on a 16-byte boundary.** This buffer must also be initialized with the correct constants for SHA-256: (0x6a09e667, 0xbb67ae85, 0x3c6ef372, 0xa54ff53a, 0x510e527f, 0x9b05688c, 0x1f83d9ab, 0x5be0cd19)
12. Load the number of bytes to be hashed into ECX.
13. Execute the REP XSHA256 instruction.

CHAPTER

3

PROGRAMMING DETAILS

3.1 MEMORY FORMATS

Only PadLock Hash Engine (PHE) data stores must be 16-byte aligned. Like other Padlock functions, PHE uses SSE datapaths and logic, and most loads to and stores from Padlock registers must be 16-byte aligned. This is true for the initial constants and resulting hash pointed at by EDI (see section 3.3)

In the Secure Hash Algorithm standard, as defined by the FIPS 180-2 specification, the input stream is defined as a sequence of 32-bit words in big-endian format. PHE is designed according to this big endian definition. However, VIA processors conform to the memory format of Intel x86 processors, which is little endian.

Therefore, the microcode of PHE automatically converts the input stream from little endian to big endian before loading PHE registers. As this process is equivalent to a 32-bit BSWAP instruction, data is loaded from memory by the XSHA instructions in 32-bit chunks, which makes the 16-byte alignment a superfluous requirement for the input stream.

The FIPS 180-2 standard actually specifies that the input is a bit stream, not a byte stream. However, like most hardware implementations of the Secure Hash Algorithm of which we are aware, PHE operates at the byte level, that is the length L of the input bit stream must be such that: L mod 8 = 0

3.2 PADDING

The FIPS 180-2 specified pre-processing of the input stream, appending the single “1” bit followed by a number of zero bits and finally the size (in bits) of the input stream, is handled automatically by the microcode for XSHA instructions. Not even the conventional “0” byte following a C-language string is required by XSHA instructions for correct processing.

3.3 DATA REGISTERS

INPUT DATA POINTER: ESI

ESI (or SI) in all XSHA instructions contains the effective address (offset) of the input data. This address is always relative to the segment specified in ES.

At the end of instruction execution, or if an interrupt (or page fault, etc.) has occurred, ESI (or SI) is incremented by the number of bytes hashed. The address is always incremented: EFLAGS.DF has no effect. Whether SI or ESI is used and updated is based on the effective address size for the executed instruction.

The padding specified by the FIPS 180-2 standard for SHA-1 and SHA-256 is performed by the microcode associated with XHSA instructions. No special termination character for the input data is required – it is perfectly valid to hash any stream of bytes including an arbitrary number of NULL (0x00) bytes.

HASH POINTER: EDI

EDI (or DI) in all XSHA instructions contains the effective address (offset) of the result data generated by the instruction. This address is always relative to the segment specified in ES. The resulting linear address must be aligned on a 128-bit boundary (16-byte aligned); otherwise a General Protection exception occurs.

*The total memory allocated at ES:EDI (or ES:DI) must be **128 bytes***. XSHA microcode requires a 16-byte aligned buffer to load the PHE hardware. Only the hash pointer at EDI is guaranteed to be aligned.

At the start of an XSHA instruction, the memory pointed to by EDI (or DI) *must contain the initial hash constants* as specified by FIPS 180-2.

For SHA-1:

```
ES:[EDI]    = 0x67452301;
ES:[EDI+4]  = 0xEFCDAB89;
ES:[EDI+8]  = 0x98BADCFE;
ES:[EDI+12] = 0x10325476;
ES:[EDI+16] = 0xC3D2E1F0;
```

For SHA-256:

```
ES:[EDI]    = 0x6A09E667;
ES:[EDI+4]  = 0xBB67AE85;
ES:[EDI+8]  = 0x3C6EF372;
ES:[EDI+12] = 0xA54FF53A;
ES:[EDI+16] = 0x510E527F;
ES:[EDI+20] = 0x9B05688C;
ES:[EDI+24] = 0x1F83D9AB;
ES:[EDI+28] = 0x5BE0CD19;
```

Note: these integers are in standard Intel little-endian format.

After each 64-byte block of the input has been processed, the values at EDI (or DI) are written to from the PHE output registers with the current hash. At this point, with the pointer in ESI incremented and the value in EAX adjusted to reflect the total number of bytes processed, the XSHA instruction is interrupt and task-switch safe.

IMPORTANT NOTE: When the current (or final) hash result is stored into memory at ES:EDI, it is stored as a series of 32-bit integers in standard Intel little-endian format. However, correct representation of the hash, as specified by FIPS 180-2, is in big-endian format. Thus the calling program will need to BSWAP each 32-bit DWORD of the hash to produce the correct result if it is to be communicated to any other program, process, or user.

Please read section 3.6 below for a more detailed discussion of endian-ness.

TOTAL NUMBER OF BYTES TO HASH: ECX

ECX contains the count of 8-bit data blocks (bytes) to be processed.

Unlike other x86 REP instructions, the value in ECX is never changed. But, also unlike other x86 REP instructions, it is a valid operation to take the hash of a null string (ECX = 0). So PadLock Hash Engine uses a second counter register (EAX) to keep track of instruction progress.

BYTES HASHED SO FAR: EAX

EAX must be initialized to 0x00000000 at the beginning of an XSHA instruction. As 64 byte blocks are hashed, the value in EAX is incremented to reflect the state of the XSHA instruction, so that interrupts and task switches can be transparently supported. When the XSHA instruction completes, EAX == ECX.

3.4 REP PREFIX

The REP XSHA instructions have similarities with the x86 REP MOVS instruction:

- The number of source bytes processed (hashed) is added to the pointer in ESI.
- An address-size prefix affects the size of ESI and EDI used.
- The usual address exceptions (past the segment limit, page fault, etc.) can occur.

And there are some different semantics:

- The value in EDI is never modified.
- The value in ECX is never modified.
- A REPNE prefix generates undefined behavior.
- An operand size prefix causes an Invalid Instruction exception.
- The DF (direction flag) in EFLAGS has no effect. The operation always proceeds from low address to high address.

3.5 EXCEPTION HANDLING

During execution of an XSHA instruction, any number of exceptions or interrupts may occur.

For most exceptions, microcode instructions appear, logically, just like atomic x86 instructions: that is, the exception or interrupt occurs only before the instruction begins, or after it ends.

For normal x86 REP instructions, these exceptions occur “naturally” when the base instruction being repeated has completed, and after all architecture registers such as source and data pointers, and the REP

counter have been updated. XSHA instructions behave the same way with respect to these controlled exceptions and interrupts.

However some exceptions occur “in the middle of” the microcode sequence. A good example is a page fault that occurs when loading a text block from ES:[ESI]. From the perspective of the XSHA instruction, the page fault occurs “instantaneously”. Execution branches to some fault handler, and, almost always, the original XSHA instruction will be re-issued by the translator.

Thus the microcode must not change an architecturally visible register until it is guaranteed that no page fault or similar exception can occur, which would cause an incorrect state on instruction restart. This is handled by not changing the x86 registers until after the XHSA microcode has issued a store instruction that has entered the MMX queue. On VIA processors, that store instruction is guaranteed to complete, and so only then are the x86 visible registers updated to reflect the state of the XSHA instruction at the completion of the store.

3.6 ON BIG- AND LITTLE- ENDIAN FORMATS

As discussed earlier in this chapter, the Secure Hash Algorithms as specified in FIPS180-2 operate on unsigned 32-bit integers in big-endian format, which conflicts with the little-endian memory format of Intel x86 and compatible processors, such as the VIA Esther processor. You can guess, correctly, that this is an annoyance and has caused no end of difficulty for cryptographers.

In the context of PHE, the difficulty occurs when dealing with the resulting hash value and the initial constants loaded into memory at the address specified in ES:EDI (or ES:DI). The perceptive reader will have noticed that PHE XSHA instructions require that the initial constants be stored in memory as the correct unsigned 32 bit integers, but in little-endian format rather than in big-endian format. This puts the data into the correct memory format for direct loading into the appropriate PHE registers.

This initialization technique causes no particular problems, and is in fact also used by most software SHA algorithm implementations. Typically these constants will be set by a compiler, and they may as well be organized in the most efficient byte order for the user’s hardware.

With respect to storing the resulting hash at ES:EDI (or ES:DI), this little-endian/big-endian issue would be no problem for PHE ***if the XSHA instructions were atomic***. There is no particular difficulty in storing the result of the hash, as unsigned 32-bit integers, in big-endian format, little-endian format, or any other format specified by the defining authority. There would be a very small, probably not detectable, increase in the time to execute an XSHA instruction as the microcode waits for the PHE output registers to be stored into memory (in little-endian format) and then does the BSWAP to make things simple for the x86 programmer.

However, XSHA instructions are not atomic. For very large input a single XSHA instruction may execute for tens or hundreds of millions of clocks. To run in an un-interruptible state for that length of time is not acceptable behavior.

Thus, after every 64-byte block of input data has been hashed, an XSHA instruction stores the current value of the hash directly to memory at ES:EDI, without any bit manipulation, in the hardware’s internal little-endian format. The bottom line is that the value of the hash (or the initial constants) must be in the same endian format when loaded at the start of an XSHA instruction, and when stored at completion of the instruction (or when the intermediate hash value is stored after each 64-byte input block).

If the unsigned 32-bit integers are stored in memory in little-endian format, the microcode and hardware of the PHE can operate at maximum efficiency. The setting of the initial constants is fundamentally a matter for the compiler, and has no performance implications. And adjusting the endian-ness of the final hash value, a few BSWAP instructions, can be performed as efficiently by the surrounding x86 code as by the microcode or hardware of PHE.

However, if the 32-bit integers are stored in the FIPS 180-2 big-endian format, it will be necessary for PHE to convert the values from big-endian to little-endian for every 64-byte data block when the results are stored, as well as during the initial load when the instruction starts. Not only that; in addition to the extra time involved in swapping the bits, this adds numerous stalls to the microcode pipeline. And makes the microcode significantly larger.

Alternately, it was (I suppose - Damn it Jim: I'm a programmer not a circuit engineer) possible to cross the wires in the hardware to handle little-endian to big-endian issues directly. But the reader will recall that PHE, like PadLock ACE, uses already existing SSE datapaths and buses, which would require not just crossing the existing wires but lots of extra transistors and a more complex, and confusing, hardware design.

It was thus a "no-brainer" for Centaur Technology's engineers to design PHE as you, the reader, now see it.

CHAPTER

4

USING PADLOCK HASH ENGINE

4.1 ENABLING

To use VIA Padlock Hash Engine (PHE), three conditions must be met:

- PHE must physically exist on the chip. Its existence can be discovered from the Centaur Extended CPUID Feature Flags. This condition is permanently set as part of the manufacturing process.
- PHE must be enabled. The enabled state can be also discovered from the Centaur Extended CPUID Feature Flags. PHE can be enabled/disabled by writing to the Function Control Register (FCR), MSR 0x1107. The RESET default is enabled.
- SSE instructions must be enabled via the standard x86 method of enabling the FXSAVE/FXRSTOR instructions using CR4[9] This CR4 enabling also enables the full set of SSE instructions on VIA Esther and Nehemiah processors.

VIA Esther and Nehemiah processors support the Centaur Extended Feature Flags. When the CPUID instruction is executed with EAX=0xC0000000, the processor returns 0xC0000001 in EAX. Note that earlier VIA C3 processors do not support the Centaur Extended CPUID Functions and will return EAX=0xC0000000 or EAX=0x00000000 (see relevant VIA processor datasheets for details on the extended CPUID function).

If a CPUID with EAX=0xC0000000 returns a value in EAX >= 0xC0000001 then Centaur Extended Feature Flags are supported. A CPUID with EAX=0xC0000001 then returns the Centaur Extended Feature Flags in EDX. There are two bits in EDX that describe PHE:

- EDX[8] == 0 PHE does not exist on this chip. FCR[28] cannot be set to enable PHE and XSHA instructions cause an Invalid Opcode Fault.
- EDX[8] == 1 PHE exists. The behavior of XSHA instructions is dependent upon whether or not PHE is enabled.
- EDX[9] == 0 PHE is disabled. XSHA instructions cause an Invalid Opcode Fault. If PHE is present, FCR[28] can be set to enable PHE.
- EDX[9] == 1 PHE is enabled (usually the RESET default). XSHA instructions behave as defined in this programming guide (provided CR4[9] =1)

VIA Esther and Nehemiah processors implement a Function Control Register (FCR), MSR 0x1107, which can be written by software to enable or disable various features. One bit controls PHE - FCR[28]:

- 0 = PHE is not enabled. If EDX[8] indicates that PHE is present, a WRMSR can set this bit to 1 thus enabling PHE.
- 1 = PHE is enabled. A WRMSR can set this bit to 0 thus disabling PHE.

4.2 STARTUP

The RESET signal is an immediate asynchronous process: RESET halts operation immediately regardless of the state of the processor. As part of the RESET process, the following PHE actions are performed:

- Any in-progress XSHA instruction is immediately cancelled and undefined results may be stored.
- The FCR is reset to its default value. This may or may not directly change PHE enable status.
- The CR4 register is reset to zero. Since this resets the FXSAVE/FXRSTOR enable, (CR4[9]), PHE is indirectly disabled.

The INIT signal is an interrupt that occurs between x86 instructions: INIT never halts instructions in the middle of their execution (except for REP string instructions, where INIT behaves like INTR). The effect of INIT on PHE is:

- The CR4 register is reset to zero. Since this resets the FXSAVE/FXRSTOR enable, (CR4[9]), PHE is indirectly disabled.

4.3 MANUFACTURED STATE

PHE is tested as part of the VIA processor manufacturing process. Relative to potential fault coverage, however, this level of testing is not 100% complete. For certain sensitive applications, additional testing (by the application) may be desirable.

CHAPTER

5

INSTRUCTION TIMING

As for all x86 instructions that reference memory, the timing of XSHA instructions is non-deterministic due to cache misses, page faults, interrupts, load and store buffer stalls, SSE queue-full stalls, external DMA snoops, etc. Consistent with other published x86 instruction timings, this section ignores these variables and assumes a perfect environment: everything is in the cache, no interrupts, no snoops, no stalls, etc.

PHE hardware completes 1 round every two clocks for SHA-1, and completes 1 round every three clocks for SHA-256. The FIPS 180-2 specification defines SHA-1 as requiring 80 rounds, and SHA-256 requires 64 rounds. Thus, for very large byte streams, the asymptotic performance of the PHE *hardware* engine is:

SHA-1 3.2 bits/clock

SHA-256 2.66 bits/clock

However, the performance of XSHA instructions is somewhat less. There is overhead in getting started and in converting from little endian format into big endian format. There is overhead in performing the pre-processing (which actually happens at the end of the instruction when all the actual source bytes have been processed). There is overhead in delivering the data to the hardware engine aligned on a 16-byte boundary (as required by the hardware engine) even though the x86 instruction does not require such alignment. And there is additional overhead in transitioning from the x86 instruction stream to the microcode. For all these reasons, and because of subtle timing interlocks, the performance of XSHA instructions is less than the hardware maximum.

We are working to minimize this overhead. In future versions of VIA processors, the hardware may be able to perform 1 round in 1 clock for both SHA-1 and SHA-256. That will be a tradeoff between performance of XSHA instructions and a possible increase in the processor die size, a decision out of this author's hands. (As your advocate for fast cryptography, I will push for the faster hash engine.)

Our current estimates for the asymptotic performance:

XSHA1 2.66 bits/clock

XSHA256 2.26 bits/clock

But remember that this includes *everything* required to generate a hash of an arbitrary byte stream. And it also assumes an infinite memory bandwidth, a delightful if unrealistic assumption.

The following performance data was measured on a 1Ghz VIA Esther processor. Note that once the stream to hash is on the order of a few hundreds of bytes, the performance of the processor approaches our hardware limit and is only slightly affected by cache misses for the largest (1 MB and 10 MB) samples tested by the program.

Sample Size (bytes)	**SHA-1 (Gb/sec)**	**SHA-256 (Gb/sec)**
10^1	0.21	0.20
10^2	1.37	1.23
10^3	2.07	1.83
10^4	2.17	1.91
10^5	2.16	1.97
10^6	1.93	1.88
10^7	1.93	1.88

CHAPTER

6

SAMPLE CODE

```
/*
  PHE Test Program -- display the (SHA-1) hash of a file
    Copyright (c) 2004-2005, Centaur Technololgy, Inc.
    All rights reserved.

  Redistribution and use in source and binary forms, with or without
  modification, are permitted provided that the following conditions are met:

   *   Redistributions of source code must retain the above copyright notice,
    this list of conditions and the following disclaimer.
   *   Redistributions in binary form must reproduce the above copyright no-
tice,
    this list of conditions and the following disclaimer in the documentation
    and/or other materials provided with the distribution.
   *   Neither the name of Centaur Technology nor the names of its contributors
    may be used to endorse or promote products derived from this software
    without specific prior written permission.

  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
  AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
  IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
  ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR  CONTRIBUTORS BE
  LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
  CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
  SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
  INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
```

```
  ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
  POSSIBILITY OF SUCH DAMAGE.

*/

/*
  Usage:  phe <file>
*/

#include <stdlib.h>
#include <stdio.h>

/* Good enough for this context */
#define SHA_1(SIZE, DATA, HASH) \
     asm __volatile__ (".byte 0xF3, 0x0F, 0xA6, 0xC8\n" \
     : \
     : "c" (SIZE), "a" (0), "S" (DATA), "D" (HASH));


int main (int argc, char *argv[]) {
  int i;
  long filesize;
  char * data = NULL;

/* Make sure this is at least 16-byte aligned */
  unsigned int * hash = (unsigned int *) valloc(128);

  FILE *f = fopen(argv[1], "r");

  if (!f) {
    printf("FILE ERROR\n");
  }
  else {
    fseek(f, 0, SEEK_END);
    filesize = ftell(f);
    data = malloc(filesize);
    rewind(f);
    fread(data, 1, filesize, f);
    fclose(f);
```

```
    hash[0] = 0x67452301;
    hash[1] = 0xefcdab89;
    hash[2] = 0x98badcfe;
    hash[3] = 0x10325476;
    hash[4] = 0xc3d2e1f0;
    SHA_1(filesize, data, hash)

    printf("SHA1(%s)= ", argv[1]);
    for (i=0; i<5; i++)
      printf("%08x", hash[i]);
    printf("\n");

    free(data);
  }
  return (0);
}
```